(2)

AVF Control Number: NBS87VDEC540_1

Ada Compiler
VALIDATION SUMMARY REPORT:
Certificate Number: 871209S1.09014
Digital Equipment Corp.
VAX Ada, Version 1.5
The host environment is the VAX 8800 under VAX/VMS, Version 4.7. The target environments are the VAX 8800 (under VAX/VMS, Version 4.7) and VAXstation II (under MicroVMS, Version 4.7)

Completion of On-Site Testing:
09 Dec 1987

Prepared By:
Software Standards Validation Group
Institute for Computer Sciences and Technology
National Bureau of Standards
Building 225, Room A266
Gaithersburg, Maryland 20899

Prepared For:
Ada Joint Program Office
United States Department of Defense
Washington, D.C. 20301-3081

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (*When Data Entered*)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETEING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (*and Subtitle*) Ada Compiler Validation Summary Report: Digital Equipment Corp.,VAX ada, Version 1.5, VAX 8800 (Host), and VAX 8800 & VAXstation II (Targets).		5. TYPE OF REPORT & PERIOD COVERED 9 Dec 1987 to 9 Dec 1988 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(*s*) National Bureau of Standards, Gaithersburg, Maryland, U.S.A.		8. CONTRACT OR GRANT NUMBER(*s*)
9. PERFORMING ORGANIZATION AND ADDRESS National Bureau of Standards, Gaithersburg, Maryland, U.S.A.		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Ada Joint Program Office United States Department of Defense Washington, DC 20301-3081		12. REPORT DATE 9 December 1987 13. NUMBER OF PAGES 71 p.
14. MONITORING AGENCY NAME & ADDRESS(*If different from Controlling Office*) National Bureau of Standards, Gaithersburg, Maryland, U.S.A.		15. SECURITY CLASS (*of this report*) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (*of this Report*)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (*of the abstract entered in Block 20. If different from Report*)

UNCLASSIFIED

18. SUPPLEMENTARY NOTES

19. KEYWORDS (*Continue on reverse side if necessary and identify by block number*)

Ada Programming language, Ada Compiler Validation Summary Report, Ada Compiler Validation Capability, ACVC, Validation Testing, Ada Validation Office, AVO, Ada Validation Facility, AVF, ANSI/MIL-STD-1815A, Ada Joint Program Office, AJPO

20. ABSTRACT (*Continue on reverse side if necessary and identify by block number*)

VAX Ada, Version 1.5, Digital Equipment Corp., National Bureau of Standards, VAX 8800 (Host) under VAX/VMS, Version 4.7 and VAX 8800 (Target) under VAX/VMS, Version 4.7, and VAXstation II (Target) under MicroVMS, Version 4.7,ACVC 1.9.

DD FORM 1473 EDITION OF 1 NOV 65 IS OBSOLETE
1 JAN 73 S/N 0102-LF-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (*When Data Entered*)

Ada Compiler Validation Summary Report:

Compiler Name: VAX Ada, Version 1.5

Certificate Number: 871209S1.09014

Host: VAX 8800 under VAX/VMS, Version 4.7

Targets: VAX 8800 under VAX/VMS, Version 4.7
VAXstation II under MicroVMS, Version 4.7

Testing Completed 09 Dec 1987 Using ACVC 1.9

This report has been reviewed and is approved.

Ada Validation Facility
Dr. David K. Jefferson
Chief, Information Systems
Engineering Division
National Bureau of Standards
Gaithersburg, MD 20899

Ada Validation Organization
Dr. John F. Kramer
Institute for Defense Analyses
Alexandria, VA 22311

Ada Joint Program Office
Virginia L. Castor
Director
Department of Defense
Washington DC 20301

EXECUTIVE SUMMARY

This Validation Summary Report (VSR) summarizes the results and conclusions of validation testing performed on the VAX Ada, Version 1.5, using Version 1.9 of the Ada Compiler Validation Capability (ACVC). VAX Ada is hosted on a VAX 8800 operating under VAX/VMS, Version 4.7. Programs processed by this compiler may be executed on:

VAX 8800 under VAX/VMS, Version 4.7
VAXstation II under MicroVMS, Version 4.7

On-site testing was performed 07 Dec 1987 through 09 Dec 1987 at Nashua, NH, under the direction of the Software Standards Validation Group, Institute for Computer Sciences and Technology, National Bureau of Standards (AVF), according to Ada Validation Organization (AVO) policies and procedures. At the time of testing, version 1.9 of the ACVC comprised 3122 tests of which 25 had been withdrawn. Of the remaining tests, 89 were determined to be inapplicable to this implementation. Results for processed Class A, C, D, and E tests were examined for correct execution. Compilation listings for Class B tests were analyzed for correct diagnosis of syntax and semantic errors. Compilation and link results of Class L tests were analyzed for correct detection of errors. The remaining 3008 tests were passed. The results of validation are summarized in the following table:

RESULT	CHAPTER													TOTAL
	2	3	4	5	6	7	8	9	10	11	12	13	14	
Passed	185	553	657	245	166	98	141	326	137	36	234	3	227	3008
Failed	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Inapplicable	19	20	18	3	0	0	2	1	0	0	0	0	26	89
Withdrawn	2	13	2	0	0	1	2	0	0	0	2	1	2	25
TOTAL	206	586	677	248	166	99	145	327	137	36	236	4	255	3122

The AVF concludes that these results demonstrate acceptable conformity to ANSI/MIL-STD-1815A Ada.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

This ~~Validation Summary~~ Report ~~(VSR)~~ describes the extent to which a specific Ada compiler conforms to the Ada Standard, ANSI/MIL-STD-1815A. This report explains all technical terms used within it and thoroughly reports the results of testing this compiler using the Ada Compiler Validation Capability (ACVC). An Ada compiler must be implemented according to the Ada Standard, and any implementation-dependent features must conform to the requirements of the Ada Standard. The Ada Standard must be implemented in its entirety, and nothing can be implemented that is not in the Standard.

Even though all validated Ada compilers conform to the Ada Standard, it must be understood that some differences do exist between implementations. The Ada Standard permits some implementation dependencies--for example, the maximum length of identifiers or the maximum values of integer types. Other differences between compilers result from the characteristics of particular operating systems, hardware, or implementation strategies. All the dependencies observed during the process of testing this compiler are given in this report.

This information in this report is derived from the test results produced during validation testing. The validation process includes submitting a suite of standardized tests, the ACVC, as inputs to an Ada compiler and evaluating the results. The purpose of validating is to ensure conformity of the compiler to the Ada Standard by testing that the compiler properly implements legal language constructs and that it identifies and rejects illegal language constructs. The testing also identifies behavior that is implementation dependent but permitted by the Ada Standard. Six classes of test are used. These tests are designed to perform checks at compile time, at link time, and during execution.

1.1 PURPOSE OF THIS VALIDATION SUMMARY REPORT

This VSR documents the results of the validation testing performed on an Ada compiler. Testing was carried out for the following purposes:

> To attempt to identify any language constructs supported by the compiler that do not conform to the Ada Standard
>
> To attempt to identify any unsupported language constructs required by the Ada Standard
>
> To determine that the implementation-dependent behavior is allowed by the Ada Standard

On-site testing was conducted from 07 Dec 1987 through 09 Dec 1987 at Nashua, NH.

1.2 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the AVO may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject compiler has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from:

> Ada Information Clearinghouse
> Ada Joint Program Office
> OUSDRE
> The Pentagon, Rm 3D-139 (Fern Street)
> Washington DC 20301-3081

or from:

> Software Standards Validation Group
> Institute for Computer Sciences and Technology
> National Bureau of Standards
> Building 225, Room A266
> Gaithersburg, Maryland 20899

Questions regarding this report or the validation test results should be directed to the AVF listed above or to:

Ada Validation Organization
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311

1.3 REFERENCES

1. Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983.

2. Ada Compiler Validation Procedures and Guidelines. Ada Joint Program Office, 1 January 1987.

3. Ada Compiler Validation Capability Implementers' Guide. SofTech, Inc., December 1986.

1.4 DEFINITION OF TERMS

ACVC	The Ada Compiler Validation Capability. The set of Ada programs that tests the conformity of an Ada compiler to the Ada programming language.
Ada Commentary	An Ada Commentary contains all information relevant to the point addressed by a comment on the Ada Standard. These comments are given a unique identification number having the form AI-ddddd.
Ada Standard	ANSI/MIL-STD-1815A, February 1983.
Applicant	The agency requesting validation.
AVF	The Ada Validation Facility. In the context of this report, the AVF is responsible for conducting compiler validations according to established procedures.
AVO	The Ada Validation Organization. In the context of this, report, the AVO is responsible for establishing procedures for compiler validations.
Compiler	A processor for the Ada language. In the context of this report, a compiler is any language processor,

including cross-compilers, translators, and interpreters.

Failed test — An ACVC test for which the compiler generates a result that demonstrates nonconformity to the Ada Standard.

Host — The computer on which the compiler resides.

Inapplicable test — An ACVC test that uses features of the language that a compiler is not required to support or may legitimately support in a way other than the one expected by the test.

Language Maintenance — The Language Maintenance Panel (LMP) is a committee established by the Ada Board to recommend interpretations and Panel possible changes to the ANSI/MIL-STD for Ada.

Passed test — An ACVC test for which a compiler generates the expected result.

Target — The computer for which a compiler generates code.

Test — An Ada program that checks a compiler's conformity regarding a particular feature or a combination of features to the Ada Standard. In the context of this report, the term is used to designate a single test, which may comprise one or more files.

Withdrawn test — An ACVC test found to be incorrect and not used to check conformity to the Ada Standard. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains illegal or erroneous use of the language.

1.5 ACVC TEST CLASSES

Conformity to the Ada Standard is measured using the ACVC. The ACVC contains both legal and illegal Ada programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable, and special program units are used to report their results during execution. Class B tests are expected to produce compilation errors. Class L tests are expected to produce link errors.

Class A tests check that legal Ada programs can be successfully compiled and executed. However, no checks are performed during execution to see if the test objective had been met. For example, a Class A test checks that reserved words of another language (other than those already reserved in the Ada language) are not treated as reserved words by an

Ada compiler. A Class A test is passed if no errors are detected at compile time and the program executes to produce a PASSED message.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that every syntax or semantic error in the test is detected. A Class B test is passed if every illegal construct that it contains is detected by the compiler.

Class C tests check that legal Ada programs can be correctly compiled and executed. Each Class C test is self-checking and produces a PASSED, FAILED, or NOT APPLICABLE message indicating the result when it is executed.

Class D tests check the compilation and execution capacities of a compiler. Since there are no capacity requirements placed on a compiler by the Ada Standard for some parameters--for example, the number of identifiers permitted in a compilation or the number of units in a library--a compiler may refuse to compile a Class D test and still be a conforming compiler. Therefore, if a Class D test fails to compile because the capacity of the compiler is exceeded, the test is classified as inapplicable. If a Class D test compiles successfully, it is self-checking and produces a PASSED or FAILED message during execution.

Each Class E test is self-checking and produces a NOT APPLICABLE, PASSED, or FAILED message when it is compiled and executed. However, the Ada Standard permits an implementation to reject programs containing some features addressed by Class E tests during compilation. Therefore, a Class E test is passed by a compiler if it is compiled successfully and executes to produce a PASSED message, or if it is rejected by the compiler for an allowable reason.

Class L tests check that incomplete or illegal Ada programs involving multiple, separately compiled units are detected and not allowed to execute. Class L tests are compiled separately and execution is attempted. A Class L test passes if it is rejected at link time--that is, an attempt to execute the main program must generate an error message before any declarations in the main program or any units referenced by the main program are elaborated.

Two library units, the package REPORT and the procedure CHECK FILE, support the self-checking features of the executable tests. The package REPORT provides the mechanism by which executable tests report PASSED, FAILED, or NOT APPLICABLE results. It also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The procedure CHECK FILE is used to check the contents of text files written by some of the Class C tests for chapter 14 of the Ada Standard. The operation of these units is checked by a set of executable tests. These tests produce messages that are examined to verify that the units are

operating correctly. If these units are not operating correctly, then the validation is not attempted.

The text of the tests in the ACVC follow conventions that are intended to ensure that the tests are reasonably portable without modification. For example, the tests make use of only the basic set of 55 characters, contain lines with a maximum length of 72 characters, use small numeric values, and place features that may not be supported by all implementations in separate tests. However, some tests contain values that require the test to be customized according to implementation-specific values--for example, an illegal file name. A list of the values used for this validation is provided in Appendix C.

A compiler must correctly process each of the tests in the suite and demonstrate conformity to the Ada Standard by either meeting the pass criteria given for the test or by showing that the test is inapplicable to the implementation. The applicability of a test to an implementation is considered each time the implementation is validated. A test that is inapplicable for one validation is not necessarily inapplicable for a subsequent validation. Any test that was determined to contain an illegal language construct or an erroneous language construct is withdrawn from the ACVC and, therefore, is not used in testing a compiler. The tests withdrawn at the time of validation are given in Appendix D.

CHAPTER 2

CONFIGURATION INFORMATION

2.1 CONFIGURATION TESTED

The candidate compilation system for this validation was tested under the following configuration:

Compiler: VAX Ada, Version 1.5

ACVC Version: 1.9

Certificate Number: 871209S1.09014

Host Computer:

Machine:	VAX 8800
Operating System:	VAX/VMS, Version 4.7
Memory Size:	68Mbytes

Target Computers:

Machine:	Operating System:	Memory Size:
VAX 8800	VAX/VMS, Version 4.7	68Mbytes
VAXstation II	MicroVMS, Version 4.7	9Mbytes

Communications Network: DECnet

2.2 IMPLEMENTATION CHARACTERISTICS

One of the purposes of validating compilers is to determine the behavior of a compiler in those areas of the Ada Standard that permit implementations to differ. Class D and E tests specifically check for such implementation differences. However, tests in other classes also characterize an implementation. The tests demonstrate the following characteristics:

- Capacities.

The compiler correctly processes tests containing loop statements nested to 65 levels, block statements nested to 65 levels, and recursive procedures separately compiled as subunits nested to 17 levels. It correctly processes a compilation containing 723 variables in the same declarative part. (See test D55A03A..H (8 tests), D56001B, D64005E..G (3 tests), and D29002K.)

- Universal integer calculations.

An implementation is allowed to reject universal integer calculations having values that exceed SYSTEM.MAX_INT. This implementation processes 64 bit integer calculations. (See tests D4A002A, D4A002B, D4A004A, and D4A004B.)

- Predefined types.

This implementation supports the additional predefined types SHORT_INTEGER, LONG_FLOAT, and SHORT_SHORT_INTEGER in the package STANDARD. (See tests B86001BC and B86001D.)

- Based literals.

An implementation is allowed to reject a based literal with a value exceeding SYSTEM.MAX_INT during compilation, or it may raise NUMERIC_ERROR or CONSTRAINT_ERROR during execution. This implementation raises NUMERIC_ERROR during execution. (See test E24101A.)

- Expression evaluation.

Apparently all default initialization expressions or record components are evaluated before any value is checked to belong to a component's subtype. (See test C32117A.)

Assignments for subtypes are performed with the same precision as the base type. (See test C35712B.)

This implementation uses no extra bits for extra precision. This implementation uses all extra bits for extra range. (See test C35903A.)

Apparently NUMERIC_ERROR is raised when an integer literal

operand in a comparison or membership test is outside the range of the base type. (See test C45232A.)

Apparently NUMERIC_ERROR is raised when a literal operand in a fixed point comparison or membership test is outside the range of the base type. (See test C45252A.)

Apparently underflow is not gradual. (See tests C45524A..Z.)

- Rounding.

The method used for rounding to integer is apparently round away from zero. (See tests C46012A..Z.)

The method used for rounding to longest integer is apparently round away from zero. (See tests C46012A..Z.)

The method used for rounding to integer in static universal real expressions is apparently round away from zero. (See test C4A014A.)

- Array types.

An implementation is allowed to raise NUMERIC_ERROR or CONSTRAINT_ERROR for an array having a 'LENGTH that exceeds STANDARD.INTEGER'LAST and/or SYSTEM.MAX_INT. For this implementation:

Declaration of an array type or subtype declaration with more than SYSTEM.MAX_INT components raises NUMERIC_ERROR. (See test C36003A.)

NUMERIC_ERROR is raised when 'LENGTH is applied to an array type with INTEGER'LAST + 2 components. NUMERIC_ERROR is raised when an array type with INTEGER'LAST + 2 components is declared. (See test C36202A.)

NUMERIC_ERROR is raised when 'LENGTH is applied to an array type with SYSTEM.MAX_INT + 2 components. NUMERIC_ERROR is raised when an array type with SYSTEM.MAX_INT + 2 components is declared. (See test C36202B.)

A packed BOOLEAN array having a 'LENGTH exceeding INTEGER'LAST raises NUMERIC_ERROR when the array type is declared. (See test C52103X.)

A packed two-dimensional BOOLEAN array with more than INTEGER'LAST components raises NUMERIC_ERROR when the array type is declared. (See test C52104Y.)

A null array with one dimension of length greater than INTEGER'LAST may raise NUMERIC_ERROR or CONSTRAINT_ERROR either

when declared or assigned. Alternatively, an implementation may accept the declaration. However, lengths must match in array slice assignments. This implementation raises NUMERIC_ERROR when the array type is declared. (See test E52103Y.)

In assigning one-dimensional array types, the expression appears to be evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. In assigning two-dimensional array types, the expression does not appear to be evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

- Discriminated types.

During compilation, an implementation is allowed to either accept or reject an incomplete type with discriminants that is used in an access type definition with a compatible discriminant constraint. This implementation accepts such subtype indications. (See test E38104A.)

In assigning record types with disciminants, the expression appears to be evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

- Aggregates.

In the evaluation of a multi-dimensional aggregate, all choices appear to be evaluated before checking against the index type. (See tests C43207A and C43207B.)

In the evaluation of an aggregate containing subaggregates, all choices are evaluated before being checked for identical bounds. (See test E43212B.)

All choices are evaluated before CONSTRAINT_ERROR is raised if a bound in a nonnull range of a nonnull aggregate does not belong to an index subtype. (See test E43211B.)

- Representation clauses.

The Ada Standard does not require an implementation to support representation clauses. If a representation clause is not supported, then the implementation must reject it.

Enumeration representation clauses containing noncontiguous values for enumeration types other than character and boolean

types are supported. (See tests C35502I..J, C35502M..N, and A39005F.)

Enumeration representation clauses containing noncontiguous values for character types are supported. (See tests C35507I..J, C35507M..N, and C55B16A.)

Enumeration representation clauses for boolean types containing representational values other than (FALSE => 0, TRUE => 1) are not supported. (See tests C35508I..J and C35508M..N.)

Length clauses with SIZE specifications for enumeration types are supported. (See test A39005B.)

Length clauses with STORAGE_SIZE specifications for access types are supported. (See tests A39005C and C87B62B.)

Length clauses with STORAGE_SIZE specifications for task types are supported. (See tests A39005D and C87B62D.)

Length clauses with SMALL specifications are supported. (See tests A39005E and C87B62C.)

Length clauses with SIZE specifications for derived integer types are supported. (See test C87B62A.)

- Pragmas.

The pragma INLINE is supported for procedures. The pragma INLINE is supported for functions. (See tests LA3004A, LA3004B, EA3004C, EA3004D, CA3004E, and CA3004F.)

- Input/output.

The package SEQUENTIAL_IO can be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests AE2101C, EE2201D, and EE2201E.)

By default, the package DIRECT_IO cannot be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests EE2401D and EE2401G.)

There are strings which are illegal external file names for SEQUENTIAL_IO and DIRECT_IO. (See tests CE2102C and CE2102H.)

Mode IN_FILE is supported for SEQUENTIAL_IO. (See test CE2102D.)

Mode OUT_FILE is supported for SEQUENTIAL_IO. (See test CE2102E.)

Modes OUT_FILE and INOUT_FILE are supported for DIRECT_IO. (See tests CE2102F and CE2102J.)

Mode IN_FILE is supported for DIRECT_IO. (See test CE2102.)

RESET and DELETE are supported for SEQUENTIAL_IO and DIRECT_IO. (See tests CE2102G and CE2102K.)

Dynamic creation and deletion of files are supported for (SEQUENTIAL_IO and DIRECT_IO. (See tests CE2106A and CE2106B.)

Overwriting to a sequential file truncates the file to last element written. (See test CE2208B.)

An existing text file can be opened in OUT_FILE mode, can be created in OUT_FILE mode, and can be created IN_FILE mode. (See test EE3102C.)

By default, only one internal file can be associated with each external file for text I/O for both reading and writing. (See tests CE2110B, CE2111D, CE3111A..E (5 tests), CE3114B, and CE3115A.)

More than one internal file can be associated with each external file for sequential I/O for reading only. (See test CE2107A.)

More than one internal file can be associated with each external file for direct I/O for reading only. (See test CE2107F.)

Temporary sequential files are given names. Temporary direct files are given names. Temporary files given names are not deleted when they are closed. (See tests CE2108A and CE2108C.)

- Generics.

Generic subprogram declarations and bodies can compiled in separate compilations. (See tests CA1012A and CA2009F.)

Generic package declarations and bodies can be compiled in separate compilations. (See tests CA2009C, BC3204C, and BC3205D.)

Generic unit bodies and their subunits can be compiled in separate compilations. (See test CA3011A.)

CHAPTER 3

TEST INFORMATION

3.1 TEST RESULTS

At the time of testing, version 1.9 of the ACVC comprised 3122 tests of which 25 had been withdrawn. Of the remaining tests, 89 were determined to be inapplicable to this implementation.

The AVF concludes that the testing results demonstrate acceptable conformity to the Ada Standard.

3.2 SUMMARY OF TEST RESULTS BY CLASS

RESULT	TEST CLASS						TOTAL
	A	B	C	D	E	L	
Passed	108	1048	1773	17	16	46	3008
Failed	0	0	0	0	0	0	0
Inapplicable	2	3	82	0	2	0	89
Withdrawn	3	2	19	0	1	0	25
TOTAL	113	1053	1874	17	19	46	3122

3.3 SUMMARY OF TEST RESULTS BY CHAPTER

RESULT	CHAPTER													TOTAL
	2	3	4	5	6	7	8	9	10	11	12	13	14	
Passed	185	553	657	245	166	98	141	326	137	36	234	3	227	3008
Failed	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Inapplicable	19	20	18	3	0	0	2	1	0	0	0	0	26	89
Withdrawn	2	13	2	0	0	1	2	0	0	0	2	1	2	25
TOTAL	206	586	677	248	166	99	145	327	137	36	236	4	255	3122

3.4 WITHDRAWN TESTS

The following 25 tests were withdrawn from ACVC Version 1.9 at the time of this validation:

B28003A	E28005C	C34004A	C35502P	A35902C	C35904A
C35A03E	C35A03R	C37213H	C37213J	C37215C	C37215E
C37215G	C37215H	C38102C	C41402A	C45614C	A74106C
C85018B	C87B04B	CC1311B	BC3105A	AD1A01A	CE2401H
CE3208A					

See Appendix D for the reason that each of these tests was withdrawn.

3.5 INAPPLICABLE TESTS

Some tests do not apply to all compilers because they make use of features that a compiler is not required by the Ada Standard to support. Others may depend on the result of another test that is either inapplicable or withdrawn. The applicability of a test to an implementation is considered each time a validation is attempted. A test that is inapplicable for one validation attempt is not necessarily inapplicable for a subsequent attempt. For this validation attempt, 89 test were inapplicable for the reasons indicated:

C24113H..Y (18 tests) have source lines that exceed the VAX Ada implementation limit of 120 characters.

A28004A Line 23 contains a pragma INTERFACE for function MEMORY_SIZE whose body is declared at line 18; this implementation rejects the subprogram body on the basis of the Ada Standard 13.9 (3). The test expects the pragma to be ignored due to the language name "ZZZZZZ". The AVO temporarily ruled this test N.A. while the issue is further considered.

C35508I..J (2 tests) and C35508M..N (2 tests) use enumeration representation clauses for boolean types containing representational values other than (FALSE => 0, TRUE => 1). These clauses are not supported by this compiler.

C35702A (and B86001CP which is not included in the above 89 count) use SHORT_FLOAT which is not supported by this implementation.

A39005G specifies a range for a component in a record representation clause that is not compatible with the default representation chosen by the compiler for the type of the component.

The following (14) tests use LONG_INTEGER, which is not supported by this compiler.

C45231C	C45304C	C45502C	C45503C	C45504C	C45504F
C45611C	C45613C	C45631C	C45632C	B52004D	B55B09C
C55B07A	B86001CS				

The following (22) tests use particular fixed point base types which are not supported by this compiler.

C35902D	C35A03B..C	C35A03O..P	C35A04B..C	C35A04O..P
C35A06B	C35A07B..C	C35A07O..P	C45531I..J	C45513M..P
C45532I..J	C45532M..P			

C86001F redefines package SYSTEM, but TEXT_IO is made obsolete by this new definition in this implementation and the test cannot be executed since the package REPORT is dependent on the package TEXT_IO.

C96005B requires the range of type DURATION to be different from those of its base type; in this implementation they are the same.

CE2102E is inapplicable because this implementation supports mode OUT_FILE for SEQUENTIAL_IO.

CE2102F is inapplicable because this implementation supports mode INOUT_FILE for DIRECT_IO.

CE2102G is inapplicable because this implementation supports RESET for SEQUENTIAL_IO.

CE2102J is inapplicable because this implementation supports mode OUT_FILE for DIRECT_IO.

CE2102K is inapplicable because this implementation supports RESET for DIRECT_IO.

CE2105A, CE2105B, CE2111H, and CE3109A are inapplicable because this implementation does not allow the creation of a file of mode IN_FILE.

CE2107B..E (4 tests), CE2107G..I (3 tests), CE2110B, CE2111D, CE3111B..E (4 tests), CE3114B, and CE3115A are inapplicable because this implementation does not allow more than one internal file to be associated with an external file for mode INOUT_FILE or OUT_FILE in combination with mode IN_FILE or OUT_FILE or INOUT_FILE when default options are used.

EE2401D and EE2401G use instantiations of package DIRECT_IO with unconstrained array types and record types having discriminants without defaults. These instantiations compiled with no errors, but during execution USE_ERROR was raised.

3.6 TEST, PROCESSING, AND EVALUATION MODIFICATIONS

It is expected that some tests will require modifications of code, processing, or evaluation in order to compensate for legitimate implementation behavior. Modifications are made with the approval of the AVO, and are made in cases where legitimate implementation behavior prevents the successful completion of an (otherwise) applicable test. Examples of such modifications include: adding a length clause to alter the default size of a collection; splitting a Class B test into sub-tests so that all errors are detected; and confirming that messages produced by an executable test demonstrate conforming behavior that wasn't anticipated by the test (such as raising one exception instead of another).

No modifications were required for any of the tests.

C34007A, C34007D, C34007G, C34007M, C34007P, and C34007S require that the attribute STORAGE_SIZE return a value greater than 1 when applied to an access subtype for which no STORAGE_SIZE length clause was provided. This requirement is challenged and will be reviewed by the ARG. The AVF verified that the failure of these tests was solely attributable to the STORAGE_SIZE check, and the AVO ruled that such results should be counted as "PASSED".

C4A012B checks that 0.0 raised to a negative power raises CONSTRAINT_ERROR; however, NUMERIC_ERROR may also be raised, and that is what this implementation does. The AVF confirmed this by an analysis of the results, and the AVO ruled that such behavior counts as "PASSED".

3.7 ADDITIONAL TESTING INFORMATION

3.7.1 Prevalidation

Prior to validation, a set of test results for ACVC Version 1.9 produced by VAX Ada was submitted to the AVF by the applicant for review. Analysis of these results demonstrated that the compiler successfully passed all applicable tests, and the compiler exhibited the expected behavior on all inapplicable tests.

3.7.2 Test Method

Testing of VAX Ada using ACVC Version 1.9 was conducted on-site by a validation team from the AVF. The configuration consisted of a VAX 8800 operating under VAX/VMS, Version 4.7 and the two target computers: VAX 8800 host operating under VAX/VMS, Version 4.7 and a VAXstation II under MicroVMS, Version 4.7. The host and target computers were linked via DECnet.

A magnetic tape containing all tests except for withdrawn tests was taken on-site by the validation team for processing. Tests that make use of implementation-specific values were customized before being written to the magnetic tape. Tests requiring modifications during the prevalidation testing were included in their modified form on the magnetic tape. The contents of the magnetic tape were loaded directly onto the host computer.

After the test files were loaded to disk, the full set of tests was compiled and linked on the VAX 8800, and all executable tests were run on the VAX 8800 and the VAXstation. Results were printed from the host computer, with results being transferred to the host computer via DECnet.

The compiler was tested using command scripts provided by Digital Equipment Corporation and reviewed by the validation team. The compiler was tested using all default (option/switch) settings except for the following:

Option/Switch	Effect
/NOCOPY_SOURCE	Controls whether the source being compiled is copied into the compilation library for a successful compilation.
/NODEBUG	Controls the inclusion of debugging symbol table information in the compiled object module.
/ERROR_LIMIT=1000	Controls the number of error level diagnostics that are allowed within a single compilation unit before the compilation is aborted.
/LIST	Controls whether a listing file is produced. /LIST without a filename uses a default filename of the form sourcename.LIS, where

	sourcename is the name of the source file being compiled.
/NOSHOW	Controls whether a portability summary is included in the listing.

Tests were compiled, linked, and executed (as appropriate) using a single host computer and two target computers. Test output, compilation listings, and job logs were captured on magnetic tape and archived at the AVF.

3.7.3 Test Site

The validation team arrived at Nashua, NH on 07 Dec 1987, and departed after testing was completed on 09 Dec 1987.

APPENDIX A

CONFORMANCE STATEMENT

The following Declaration of Conformance is provided by DEC for VAX Ada. Because the VAXELN targets produce different results than those of the VMS targets for three ACVC tests (which require temporary files to have names), the VAXELN & VMS operating environments were tested separately, and the testing is thus documented in separate VSRs. However, the AVO made no request for DEC to submit separate Declarations of Conformance.

Declaration of Conformance

Compiler Implementer: Digital Equipment Corporation
Ada Validation Facility: National Bureau of Standards
Ada Compiler Validation Capability Version: 1.9

Base Configuration:

Compiler: VAX Ada Version 1.5

Host Configuration:

VAX 8800 (under VAX/VMS, Version 4.7)

Target Configuration:

VAX 8800 (under VAX/VMS, Version 4.7)
VAXstation II (under MicroVMS, Version 4.7)
MicroVAX II (under VAXELN Toolkit, Version 3.0
in combination with VAXELN Ada, Version 1.2)

Derived Compiler Registration:

Compiler: VAX Ada Version 1.5

Host Configuration:

All members of the VAX family:
MicroVAX I
VAXstation I
MicroVAX II
VAXstation II
VAXstation 2000
(all under MicroVMS, Version 4.7)

MicroVAX 3500
MicroVAX 3600
VAXserver 3500
VAXserver 3600
VAXserver 3602
VAXstation 3200
VAXstation 3500
(all under VAX/VMS, Version 4.7A)

VAX-11/730
VAX-11/750
VAX-11/780
VAX-11/782
VAX-11/785

VAX 8200
VAX 8250
VAX 8300
VAX 8350
VAX 8500
VAX 8530
VAX 8550
VAX 8600
VAX 8650
VAX 8700
VAX 8800 (base configuration)
(all under VAX/VMS, Version 4.7)

Target Configuration:

Same as Host; and the following VAXELN configurations

MicroVAX I
MicroVAX II
rtVAX 1000
KA620 (rtVAX 1000 processor board)
MicroVAX 3500
MicroVAX 3600
VAX-11/730
VAX-11/750
VAX 8500
VAX 8530
VAX 8550
VAX 8700
VAX 8800
(all under VAXELN Toolkit, Version 3.0 in combination with VAXELN Ada, Version 1.2)

All of the processors listed above, including MicroVAX, VAXstation, and VAXserver systems, are members of the VAX family. The VAX family includes multiple hardware/software implementations of the same instruction set architecture. All processors of the VAX family together with the VMS or MicroVMS operating system provide an identical user mode instruction set execution environment and need not be distinguished for purposes of validation. Similarly, all VAX family processors supported as VAXELN Toolkit targets provide an identical user mode instruction set execution environment.

The identical VAX Ada compiler is used on all hosts, and the compiler has no knowledge of the particular VAX model on which it is being executed. Further, the compiler generates identical code for all targets. Thus, the code generated on any VAX host can be executed without modification on any of the VAX targets listed above.

All of the configurations listed under the derived compiler registration section above are equivalent to the base configuration. That is, all applicable ACVC Version 1.9 tests could be correctly compiled and executed on any of the configurations listed.

14 September 1987

William J. Heffner
Vice President, System Software Group

APPENDIX B

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in chapter 13 of MIL-STD-1815A, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of the VAX Ada, Version 1.5, are described in the following sections which discuss topics in Appendix F of the Ada Language Reference Manual (ANSI/MIL-STD-1815A).. Implementation-specific portions of the package STANDARD are also included in this appendix.

```
package STANDARD is


     type INTEGER is range -2147483648 .. 2147483647;
     type SHORT_INTEGER is range -32768  .. 32767 ;

     type FLOAT is digits 6 range;
     type LONG_FLOAT is digits 15;
     type LONG_LONG_FLOAT is 33 digits;

     type DURATION is delta 1.0E-4 range -131072.0  .. 131071.9999;

end STANDARD;
```

APPENDIX B

APPENDIX F OF THE ADA STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in chapter 13 of ANSI/MIL-STD-1815A-1983, and to certain allowed restrictions on representation classes. The implementation-dependent characteristics are described in the following sections which discuss topics one through eight as stated in Appendix F of the Ada Language Reference manual (ANSI/MIL-STD-1815A). Two other sections, package STANDARD and file naming conventions, are also included in this appendix.

Portions of this section refer to the following attachments:

1. Attachment 1 - Implementation-Dependent Pragmas

2. Attachment 2 - VAX Ada Appendix F

(1) Implementation-Dependent Pragmas

See Attachment 1.

(2) Implementation-Dependent Attributes

Name	Type
P'AST_ENTRY	The value of this attribute is of type SYSTEM.AST_HANDLER.
P'BIT	The value of this attribute is of type universal_integer.
P'MACHINE_SIZE	The value of this attribute is of type universal_integer.

P'NULL_PARAMETER	The value of this attribute is of type P.
P'TYPE_CLASS	The value of this attribute is of type SYSTEM.TYPE_CLASS.

(3) Package SYSTEM

See Attachment 2, Section F.3.

(4) Representation Clause Restrictions

See Attachment 2, Section F.4.

(5) Conventions

See Attachment 2, Section F.5.

(6) Address Clauses

See Attachment 2, Section F.6.

(7) Unchecked Conversions

VAX Ada supports the generic function UNCHECKED_CONVERSION with the following restrictions on the class of types involved:

1. The actual subtype corresponding to the formal type TARGET must not be an unconstrained array type.

2. The actual subtype coresponding to the formal type TARGET must not be an unconstrained type with discriminants.

(8) Input-Output Packages

SEQUENTIAL_IO Package

SEQUENTIAL_IO can be instantiated with any file type, including an unconstrained array type or an unconstrained record type. However, input-output for access types is erroneous.

VAX Ada provides full support for SEQUENTIAL_IO, with the following restrictions and clarifications:

1. VAX Ada supports modes IN_FILE and OUT_FILE for sequential input-output. However, VAX Ada does not allow the creation of a file of mode IN_FILE.

2. More than one internal file can be associated with the same external file. However, with default FORM strings, this is only allowed when all internal files have mode IN_FILE (multiple readers). If one or more internal files have mode OUT_FILE (mixed readers and writers or multiple writers), then sharing can only be achieved using FORM strings.

3. VAX Ada supports deletion of an external file which is associated with more than one internal file. In this case, the external file becomes immediately unavailable for any new associations, but the current associations are not affected; the external file is actually deleted after the last association has been broken.

4. VAX Ada allows resetting of shared files, but an implementation restriction does not allow the mode of a file to be changed from IN_FILE to OUT_FILE (an amplification of accessing privileges while the external file is being accessed).

DIRECT_IO Package

```
type CNT is range 0 .. 2147483647;
```

TEXT_IO Package

```
type CNT is range 0 .. 2147483647;
subtype FIELD is INTEGER range 0 .. 2147483647;
```

LOW_LEVEL_IO

Low-level input-output is not provided.

(9) Package STANDARD

```
type INTEGER is range -2147483648 .. 2147483647;
type SHORT_INTEGER is range -32768 .. 32767;
type SHORT_SHORT_INTEGER is range -128 .. 127;
-- type LONG_INTEGER is not supported

type FLOAT is digits 6;
type LONG_FLOAT is digits 15;
type LONG_LONG_FLOAT is digits 33;
-- type SHORT_FLOAT is not supported

type DURATION is delta 1.0E-4
                 range -131072.0 .. 131071.9999;
```

(10) File Names

File names follow the conventions and restrictions of the target operating system.

Attachment 1

Predefined Language Pragmas

1 This annex defines the pragmas LIST, PAGE, and OPTIMIZE, and summarizes the definitions given elsewhere of the remaining language-defined pragmas.

The VAX Ada pragma TITLE is also defined in this annex.

Pragma	Meaning
AST_ENTRY	Takes the simple name of a single entry as the single argument; at most one AST_ENTRY pragma is allowed for any given entry. This pragma must be used in combination with the AST_ENTRY attribute, and is only allowed after the entry declaration and in the same task type specification or single task as the entry to which it applies. This pragma specifies that the given entry may be used to handle a VAX/VMS asynchronous system trap (AST) resulting from a VAX/VMS system service call. The pragma does not affect normal use of the entry (see 9.12a).

2	CONTROLLED	Takes the simple name of an access type as the single argument. This pragma is only allowed immediately within the declarative part or package specification that contains the declaration of the access type; the declaration must occur before the pragma. This pragma is not allowed for a derived type. This pragma specifies that automatic storage reclamation must not be performed for objects designated by values of the access type, except upon leaving the innermost block statement, subprogram body, or task body that encloses the access type declaration, or after leaving the main program (see 4.8).
3	ELABORATE	Takes one or more simple names denoting library units as arguments. This pragma is only allowed immediately after the context clause of a compilation unit (before the subsequent library unit or secondary unit). Each argument must be the simple name of a library unit mentioned by the context clause. This pragma specifies that the corresponding library unit body must be elaborated before the given compilation unit. If the given compilation unit is a subunit, the library unit body must be elaborated before the body of the ancestor library unit of the subunit (see 10.5).
	EXPORT_EXCEPTION	Takes an internal name denoting an exception, and optionally takes an external designator (the name of a VAX/VMS Linker global symbol), a form (ADA or VMS), and a code (a static integer expression that is interpreted as a VAX condition code)

as arguments. A code value must be specified when the form is VMS (the default if the form is not specified). This pragma is only allowed at the place of a declarative item, and must apply to an exception declared by an earlier declarative item of the same declarative part or package specification; it is not allowed for an exception declared with a renaming declaration. This pragma permits an Ada exception to be handled by programs written in other VAX languages (see 13.9a.3.2).

EXPORT_FUNCTION

Takes an internal name denoting a function, and optionally takes an external designator (the name of a VAX/VMS Linker global symbol), parameter types, and result type as arguments. This pragma is only allowed at the place of a declarative item, and must apply to a function declared by an earlier declarative item of the same declarative part or package specification. In the case of a function declared as a compilation unit, the pragma is only allowed after the function declaration and before any subsequent compilation unit. This pragma is not allowed for a function declared with a renaming declaration, and is not allowed for a generic function (it may be given for a generic instantiation). This pragma permits an Ada function to be called from a program written in another VAX language (see 13.9a.1.4).

EXPORT_OBJECT

Takes an internal name denoting an object, and optionally takes an external designator (the name of a VAX/VMS Linker global symbol) and size designator (a VAX/VMS

Linker global symbol whose value is the size in bytes of the exported object) as arguments. This pragma is only allowed at the place of a declarative item at the outermost level of a library package specification or body, and must apply to a variable declared by an earlier declarative item of the same package specification or body; the variable must be of a type or subtype that has a constant size at compile time. This pragma is not allowed for objects declared with a renaming declaration, and is not allowed in a generic unit. This pragma permits an Ada object to be referred to by a routine written in another VAX language (see 13.9a.2.2).

EXPORT_PROCEDURE

Takes an internal name denoting a procedure, and optionally takes an external designator (the name of a VAX/VMS Linker global symbol) and parameter types as arguments. This pragma is only allowed at the place of a declarative item, and must apply to a procedure declared by an earlier declarative item of the same declarative part or package specification. In the case of a procedure declared as a compilation unit, the pragma is only allowed after the procedure declaration and before any subsequent compilation unit. This pragma is not allowed for a procedure declared with a renaming declaration, and is not allowed for a generic procedure (it may be given for a generic instantiation). This pragma permits an Ada routine to be called from a program

written in another VAX language (see 13.9a.1.4).

EXPORT_VALUED_PROCEDURE	Takes an internal name denoting a procedure, and optionally takes an external designator (the name of a VAX/VMS Linker global symbol) and parameter types as arguments. This pragma is only allowed at the place of a declarative item, and must apply to a procedure declared by an earlier declarative item of the same declarative part or package specification. In the case of a procedure declared as a compilation unit, the pragma is only allowed after the procedure declaration and before any subsequent compilation unit. The first (or only) parameter of the procedure must be of mode out. This pragma is not allowed for a procedure declared with a renaming declaration and is not allowed for a generic procedure (it may be given for a generic instantiation). This pragma permits an Ada procedure to behave as a function that both returns a value and causes side effects on its parameters when it is called from a routine written in another VAX language (see 13.9a.1.4).
IMPORT_EXCEPTION	Takes an internal name denoting an exception, and optionally takes an external designator (the name of a VAX/VMS Linker global symbol), a form (ADA or VMS), and a code (a static integer expression that is interpreted as a VAX condition code) as arguments. A code value is allowed only when the form is VMS (the default if the form is not specified). This pragma

is only allowed at the place of a declarative item, and must apply to an exception declared by an earlier declarative item of the same declarative part or package specification; it is not allowed for an exception declared with a renaming declaration. This pragma permits a non-Ada exception (most notably, a VAX condition) to be handled by an Ada program (see 13.9a.3.1).

IMPORT_FUNCTION

Takes an internal name denoting a function, and optionally takes an external designator (the name of a VAX/VMS Linker global symbol), parameter types, result type, and mechanism as arguments. Pragma INTERFACE must be used with this pragma (see 13.9). This pragma is only allowed at the place of a declarative item, and must apply to a function declared by an earlier declarative item of the same declarative part or package specification. In the case of a function declared as a compilation unit, the pragma is only allowed after the function declaration and before any subsequent *compilation* unit. This pragma is allowed for a function declared with a renaming declaration; it is not allowed for a generic function or a generic function instantiation. This pragma permits a non-Ada routine to be used as an Ada function (see 13.9a.1.1).

IMPORT_OBJECT

Takes an internal name denoting an object, and optionally takes an external designator (the name of a VAX/VMS Linker global symbol) and size (a VAX/VMS Linker global symbol whose value is the size in bytes of the imported object) as arguments. This pragma is only

allowed at the place of a declarative item at the outermost level of a library package specification or body, and must apply to a variable declared by an earlier declarative item of the same package specification or body; the variable must be of a type or subtype that has a constant size at compile time. This pragma is not allowed for objects declared with a renaming declaration, and is not allowed in a generic unit. This pragma permits storage declared in a non-Ada routine to be referred to by an Ada program (see 13.9a.2.1).

IMPORT_PROCEDURE	Takes an internal name denoting a procedure, and optionally takes an external designator (the name of a VAX/VMS Linker global symbol) parameter types, and mechanism as arguments. Pragma INTERFACE must be used with this pragma (see 13.9). This pragma is only allowed at the place of a declarative item, and must apply to a procedure declared by an earlier declarative item of the same declarative part or package specification. In the case of a procedure declared as a compilation unit, the pragma is only allowed after the procedure declaration and before any subsequent compilation unit. This pragma is allowed for a procedure declared with a renaming declaration; it is not allowed for a generic procedure or a generic procedure instantiation. This pragma permits a non-Ada routine to be used as an Ada procedure (see 13.9a.1.1).
IMPORT_VALUED_PROCEDURE	Takes an internal name denoting a procedure, and optionally takes an external designator (the name of a

VAX/VMS Linker global symbol), parameter types, and mechanism as arguments. Pragma INTERFACE must be used with this pragma (see 13.9). This pragma is only allowed at the place of a declarative item, and must apply to a procedure declared by an earlier declarative item of the same declarative part or package specification. In the case of a procedure declared as a compilation unit, the pragma is only allowed after the procedure declaration and before any subsequent compilation unit. The first (or only) parameter of the procedure must be of mode out. This pragma is allowed for a procedure declared with a renaming declaration; it is not allowed for a generic procedure. This pragma permits a non-Ada routine that returns a value and causes side effects on its parameters to be used as an Ada procedure (see 13.9a.1.1).

4	INLINE	Takes one or more names as arguments; each name is either the name of a subprogram or the name of a generic subprogram. This pragma is only allowed at the place of a declarative item in a declarative part or package specification, or after a library unit in a compilation, but before any subsequent compilation unit. This pragma specifies that the subprogram bodies should be expanded inline at each call whenever possible; in the case of a generic subprogram, the pragma applies to calls of its instantiations (see 6.3.2).
5	INTERFACE	Takes a language name and a subprogram name as arguments. This

pragma is allowed at the place of a declarative item, and must apply in this case to a subprogram declared by an earlier declarative item of the same declarative part or package specification. This pragma is also allowed for a library unit; in this case the pragma must appear after the subprogram declaration, and before any subsequent compilation unit. This pragma specifies the other language (and thereby the calling conventions) and informs the compiler that an object module will be supplied for the corresponding subprogram (see 13.9).

In VAX Ada, pragma INTERFACE is required in combination with pragmas IMPORT_FUNCTION, IMPORT_PROCEDURE, and IMPORT_VALUED_PROCEDURE (see 13.9a.1).

6 LIST

Takes one of the identifiers ON or OFF as the single argument. This pragma is allowed anywhere a pragma is allowed. It specifies that listing of the compilation is to be continued or suspended until a LIST pragma with the opposite argument is given within the same compilation. The pragma itself is always listed if the compiler is producing a listing.

LONG_FLOAT

Takes either D_FLOAT or G_FLOAT as the single argument. The default is G_FLOAT. This pragma is only allowed at the start of a compilation, before the first compilation unit (if any) of the

compilation. It specifies the choice of representation to be used for the predefined type LONG_FLOAT in package STANDARD and for floating point type declarations with digits specified in the range 7..15 (see 3.5.7a).

MAIN_STORAGE

Takes one or two nonnegative static simple expressions of some integer type as arguments. This pragma is only allowed in the outermost declarative part of a library subprogram; at most one such pragma is allowed in a library subprogram. It has an effect only when the subprogram to which it applies is used as a main program. This pragma causes a fixed-size stack to be created for a main task (the task associated with a main program), and determines the number of storage units (bytes) to be allocated for the stack working storage area or guard pages or both. The value specified for either or both the working storage area and guard pages is rounded up to an integral number of pages. A value of zero for the working storage area results in the use of a default size; a value of zero for the guard pages results in no guard storage. A negative value for either working storage or guard pages causes the pragma to be ignored (see 13.2b).

7 MEMORY_SIZE

Takes a numeric literal as the single argument. This pragma is only allowed at the start of a compilation, before the first compilation unit (if any) of the

compilation. The effect of this pragma is to use the value of the specified numeric literal for the definition of the named number MEMORY_SIZE (see 13.7).

8	OPTIMIZE	Takes one of the identifiers TIME or SPACE as the single argument. This pragma is only allowed within a declarative part and it applies to the block or body enclosing the declarative part. It specifies whether time or space is the primary optimization criterion.
		In VAX Ada, this pragma is only allowed immediately within a declarative part of a body declaration.
9	PACK	Takes the simple name of a record or array type as the single argument. The allowed positions for this pragma, and the restrictions on the named type, are governed by the same rules as for a representation clause. The pragma specifies that storage minimization should be the main criterion when selecting the representation of the given type (see 13.1).
10	PAGE	This pragma has no argument, and is allowed anywhere a pragma is allowed. It specifies that the program text which follows the pragma should start on a new page (if the compiler is currently producing a listing).
11	PRIORITY	Takes a static expression of the predefined integer subtype PRIORITY as the single argument. This pragma is only allowed within the specification of a task unit or

immediately within the outermost declarative part of a main program. It specifies the priority of the task (or tasks of the task type) or the priority of the main program (see 9.8).

PSECT_OBJECT

Takes an internal name denoting an object, and optionally takes an external designator (the name of a program section) and a size (a VAX/VMS Linker global symbol whose value is interpreted as the size in bytes of the exported /imported object) as arguments. This pragma is only allowed at the place of a declarative item at the outermost level of a library package specification or body, and must apply to a variable declared by an earlier declarative item of the same package specification or body; the variable must be of a type or subtype that has a constant size at compile time. This pragma is not allowed for an object declared with a renaming declaration, and is not allowed in a generic unit. This pragma enables the shared use of objects that are stored in overlaid program sections (see 13.9a.2.3).

12 SHARED

Takes the simple name of a variable as the single argument. This pragma is allowed only for a variable declared by an object declaration and whose type is a scalar or access type; the variable declaration and the pragma must both occur (in this order) immediately within the same declarative part or package specification. This pragma specifies that every read or update

of the variable is a synchronization point for that variable. An implementation must restrict the objects for which this pragma is allowed to objects for which each of direct reading and direct updating is implemented as an indivisible operation (see 9.11).

VAX Ada does not support pragma SHARED (see VOLATILE).

13	STORAGE_UNIT	Takes a numeric literal as the single argument. This pragma is only allowed at the start of a compilation, before the first compilation unit (if any) of the compilation. The effect of this pragma is to use the value of the specified numeric literal for the definition of the named number STORAGE_UNIT (see 13.7).

In VAX Ada, the only argument allowed for this pragma is eight (8).

14	SUPPRESS	Takes as arguments the identifier of a check and optionally also the name of either an object, a type or subtype, a subprogram, a task unit, or a generic unit. This pragma is only allowed either immediately within a declarative part or immediately within a package specification. In the latter case, the only allowed form is with a name that denotes an entity (or several overloaded subprograms) declared immediately within the package specification. The permission to omit the given check extends from the place of the pragma to the end of the declarative region associated

with the innermost enclosing block statement or program unit. For a pragma given in a package specification, the permission extends to the end of the scope of the named entity.

If the pragma includes a name, the permission to omit the given check is further restricted: it is given only for operations on the named object or on all objects of the base type of a named type or subtype; for calls of a named subprogram; for activations of tasks of the named task type; or for instantiations of the given generic unit (see 11.7).

VAX Ada does not support pragma SUPPRESS (see SUPPRESS_ALL).

SUPPRESS_ALL — This pragma has no argument and is only allowed following a compilation unit. This pragma specifies that all run-time checks in the unit are suppressed (see 11.7).

15 SYSTEM_NAME — Takes an enumeration literal as the single argument. This pragma is only allowed at the start of a compilation, before the first compilation unit (if any) of the compilation. The effect of this pragma is to use the enumeration literal with the specified identifier for the definition of the constant SYSTEM_NAME. This pragma is only allowed if the specified identifier corresponds to one of the literals of the type NAME declared in the package SYSTEM (see 13.7).

TASK_STORAGE	Takes the simple name of a task type and a static expression of some integer type as arguments. This pragma is allowed anywhere that a task storage specification is allowed; that is, the declaration of the task type to which the pragma applies and the pragma must both occur (in this order) immediately within the same declarative part, package specification, or task specification. The effect of this pragma is to use the value of the expression as the number of storage units (bytes) to be allocated as guard storage. The value is rounded up to an integral number of pages: a value of zero results in no guard storage; a negative value causes the pragma to be ignored (see 13.2a).
TIME_SLICE	Takes a static expression of the predefined fixed point type DURATION (in package STANDARD) as the single argument. This pragma is only allowed in the outermost declarative part of a library subprogram, and at most one such pragma is allowed in a library subprogram. It has an effect only when the subprogram to which it applies is used as a main program. This pragma specifies the nominal amount of elapsed time permitted for the execution of a task when other tasks of the same priority are also eligible for execution. A positive, nonzero value of the static expression enables round-robin scheduling for all tasks in the subprogram; a negative or zero value disables it (see 9.8a).

TITLE

Takes a title or a subtitle string, or both, in either order, as arguments. Pragma TITLE has the form:

```
pragma TITLE (titling-option
   [,titling-option]);

titling-option :=
     [TITLE =>] string_literal
   | [SUBTITLE =>] string_literal
```

This pragma is allowed anywhere a pragma is allowed; the given strings supersedes the default title and/or subtitle portions of a compilation listing.

VOLATILE

Takes the simple name of a variable as the single argument. This pragma is only allowed for a variable declared by an object declaration. The variable declaration and the pragma must both occur (in this order) immediately within the same declarative part or package specification. The pragma must appear before any occurrence of the name of the variable other than in an address clause or in one of the VAX Ada pragmas IMPORT_OBJECT, EXPORT_OBJECT, or PSECT_OBJECT. The variable cannot be declared by a renaming declaration. The VOLATILE pragma specifies that the variable may be modified asynchronously. This pragma instructs the compiler to obtain the value of a variable from memory each time it is used (see 9.11).

Attachment 2

Implementation-Dependent Characteristics

NOTE

This appendix is not part of the standard definition of the Ada programming language.

This appendix summarizes the implementation-dependent characteristics of VAX Ada by

- Listing the VAX Ada pragmas and attributes.
- Giving the specification of the package SYSTEM.
- Presenting the restrictions on representation clauses and unchecked type conversions.
- Giving the conventions for names denoting implementation-dependent components in record representation clauses.
- Giving the interpretation of expressions in address clauses.
- Presenting the implementation-dependent characteristics of the input-output packages.
- Presenting other implementation-dependent characteristics.

F.1 Implementation-Dependent Pragmas

VAX Ada provides the following pragmas, which are defined elsewhere in the text. In addition, VAX Ada restricts the predefined language pragmas INLINE and INTERFACE, and provides alternatives to the pragmas SHARED and SUPPRESS (VOLATILE and SUPPRESS_ALL). See Annex B for a descriptive pragma summary.

- AST_ENTRY (see 9.12a)
- EXPORT_EXCEPTION (see 13.9a.3.2)
- EXPORT_FUNCTION (see 13.9a.1.4)
- EXPORT_OBJECT (see 13.9a.2.2)
- EXPORT_PROCEDURE (see 13.9a.1.4)
- EXPORT_VALUED_PROCEDURE (see 13.9a.1.4)
- IMPORT_EXCEPTION (see 13.9a.3.1)
- IMPORT_FUNCTION (see 13.9a.1.1)
- IMPORT_OBJECT (see 13.9a.2.1)
- IMPORT_PROCEDURE (see 13.9a.1.1)
- IMPORT_VALUED_PROCEDURE (see 13.9a.1.1)
- LONG_FLOAT (see 3.5.7a)
- MAIN_STORAGE (see 13.2b)
- PSECT_OBJECT (see 13.9a.2.3)
- SUPPRESS_ALL (see 11.7)
- TASK_STORAGE (see 13.2a)
- TIME_SLICE (see 9.8a)
- TITLE (see B)
- VOLATILE (see 9.11)

F.2 Implementation-Dependent Attributes

VAX Ada provides the following attributes, which are defined elsewhere in the text. See Annex A for a descriptive attribute summary.

- AST_ENTRY (see 9.12a)
- BIT (see 13.7.2)
- MACHINE_SIZE (see 13.7.2)
- NULL_PARAMETER (see 13.9a.1.3)
- TYPE_CLASS (see 13.7a.2)

F.3 Specification of the Package System

```
package SYSTEM is

    type NAME is (VAX_VMS, VAXELN);

    SYSTEM_NAME    : constant NAME := VAX_VMS;
    STORAGE_UNIT   : constant := 8;
    MEMORY_SIZE    : constant := 2**31-1;
    MAX_INT        : constant := 2**31-1;
    MIN_INT        : constant := -(2**31);
    MAX_DIGITS     : constant := 33;
    MAX_MANTISSA   : constant := 31;
    FINE_DELTA     : constant := 2.0**(-31);
    TICK           : constant := 10.0**(-2);

    subtype PRIORITY is INTEGER range 0 .. 15;

--  Address type
--
    type ADDRESS is private;

    ADDRESS_ZERO : constant ADDRESS;

    function "+" (LEFT : ADDRESS; RIGHT : INTEGER) return ADDRESS;
    function "+" (LEFT : INTEGER; RIGHT : ADDRESS) return ADDRESS;
    function "-" (LEFT : ADDRESS; RIGHT : ADDRESS) return INTEGER;
    function "-" (LEFT : ADDRESS; RIGHT : INTEGER) return ADDRESS;

--  function "="  (LEFT, RIGHT : ADDRESS) return BOOLEAN;
--  function "/=" (LEFT, RIGHT : ADDRESS) return BOOLEAN;
    function "<"  (LEFT, RIGHT : ADDRESS) return BOOLEAN;
    function "<=" (LEFT, RIGHT : ADDRESS) return BOOLEAN;
    function ">"  (LEFT, RIGHT : ADDRESS) return BOOLEAN;
    function ">=" (LEFT, RIGHT : ADDRESS) return BOOLEAN;
```

```
--  Note that because ADDRESS is a private type
--  the functions "=" and "/=" are already available and
--  do not have to be explicitly defined

    generic
        type TARGET is private;
    function FETCH_FROM_ADDRESS (A : ADDRESS) return TARGET;

    generic
        type TARGET is private;
    procedure ASSIGN_TO_ADDRESS (A : ADDRESS; T : TARGET);

    type TYPE_CLASS is (TYPE_CLASS_ENUMERATION,
                        TYPE_CLASS_INTEGER,
                        TYPE_CLASS_FIXED_POINT,
                        TYPE_CLASS_FLOATING_POINT,
                        TYPE_CLASS_ARRAY,
                        TYPE_CLASS_RECORD,
                        TYPE_CLASS_ACCESS,
                        TYPE_CLASS_TASK,
                        TYPE_CLASS_ADDRESS);

--  VAX Ada floating point type declarations for the VAX
--  hardware floating point data types

    type D_FLOAT is implementation_defined;
    type F_FLOAT is implementation_defined;
    type G_FLOAT is implementation_defined;
    type H_FLOAT is implementation_defined;

--  AST handler type

    type AST_HANDLER is limited private;

    NO_AST_HANDLER : constant AST_HANDLER;

--  Non-Ada exception

    NON_ADA_ERROR : exception;

--  VAX hardware-oriented types and functions

    type    BIT_ARRAY is array (INTEGER range <>) of BOOLEAN;
    pragma  PACK(BIT_ARRAY);

    subtype BIT_ARRAY_8 is BIT_ARRAY  (0 ..  7);
    subtype BIT_ARRAY_16 is BIT_ARRAY (0 .. 15);
    subtype BIT_ARRAY_32 is BIT_ARRAY (0 .. 31);
    subtype BIT_ARRAY_64 is BIT_ARRAY (0 .. 63);

    type UNSIGNED_BYTE  is range 0 .. 255;
    for  UNSIGNED_BYTE'SIZE  use 8;
```

```
function "not" (LEFT        : UNSIGNED_BYTE) return UNSIGNED_BYTE;
function "and" (LEFT, RIGHT : UNSIGNED_BYTE) return UNSIGNED_BYTE;
function "or"  (LEFT, RIGHT : UNSIGNED_BYTE) return UNSIGNED_BYTE;
function "xor" (LEFT, RIGHT : UNSIGNED_BYTE) return UNSIGNED_BYTE;

function TO_UNSIGNED_BYTE (X : BIT_ARRAY_8) return UNSIGNED_BYTE;
function TO_BIT_ARRAY_8 (X : UNSIGNED_BYTE) return BIT_ARRAY_8;

type UNSIGNED_BYTE_ARRAY is array (INTEGER range <>) of UNSIGNED_BYTE;

type UNSIGNED_WORD  is range 0 .. 65535
for  UNSIGNED_WORD'SIZE  use 16;

function "not" (LEFT        : UNSIGNED_WORD) return UNSIGNED_WORD;
function "and" (LEFT, RIGHT : UNSIGNED_WORD) return UNSIGNED_WORD;
function "or"  (LEFT, RIGHT : UNSIGNED_WORD) return UNSIGNED_WORD;
function "xor" (LEFT, RIGHT : UNSIGNED_WORD) return UNSIGNED_WORD;

function TO_UNSIGNED_WORD (X : BIT_ARRAY_16) return UNSIGNED_WORD;
function TO_BIT_ARRAY_16 (X : UNSIGNED_WORD) return BIT_ARRAY_16;

type UNSIGNED_WORD_ARRAY is array (INTEGER range <>) of UNSIGNED_WORD;

type UNSIGNED_LONGWORD is range MIN_INT .. MAX_INT;
for UNSIGNED_LONGWORD'SIZE use 32;

function "not" (LEFT        : UNSIGNED_LONGWORD) return UNSIGNED_LONGWORD;
function "and" (LEFT, RIGHT : UNSIGNED_LONGWORD) return UNSIGNED_LONGWORD;
function "or"  (LEFT, RIGHT : UNSIGNED_LONGWORD) return UNSIGNED_LONGWORD;
function "xor" (LEFT, RIGHT : UNSIGNED_LONGWORD) return UNSIGNED_LONGWORD;

function TO_UNSIGNED_LONGWORD (X : BIT_ARRAY_32)
   return UNSIGNED_LONGWORD;
function TO_BIT_ARRAY_32 (X : UNSIGNED_LONGWORD) return BIT_ARRAY_32;

type UNSIGNED_LONGWORD_ARRAY is
   array (INTEGER range <>) of UNSIGNED_LONGWORD;

type UNSIGNED_QUADWORD is record
    L0 : UNSIGNED_LONGWORD;
    L1 : UNSIGNED_LONGWORD;
    end record;
for UNSIGNED_QUADWORD'SIZE use 64;

function "not" (LEFT        : UNSIGNED_QUADWORD) return UNSIGNED_QUADWORD;
function "and" (LEFT, RIGHT : UNSIGNED_QUADWORD) return UNSIGNED_QUADWORD;
function "or"  (LEFT, RIGHT : UNSIGNED_QUADWORD) return UNSIGNED_QUADWORD;
function "xor" (LEFT, RIGHT : UNSIGNED_QUADWORD) return UNSIGNED_QUADWORD;

function TO_UNSIGNED_QUADWORD (X : BIT_ARRAY_64)
   return UNSIGNED_QUADWORD;
function TO_BIT_ARRAY_64 (X : UNSIGNED_QUADWORD) return BIT_ARRAY_64;
```

```
type UNSIGNED_QUADWORD_ARRAY is
   array (INTEGER range <>) of UNSIGNED_QUADWORD;

function TO_ADDRESS  (X : INTEGER)              return ADDRESS;
function TO_ADDRESS  (X : UNSIGNED_LONGWORD)    return ADDRESS;
function TO_ADDRESS  (X : universal_integer)    return ADDRESS;

function TO_INTEGER           (X : ADDRESS)     return INTEGER;
function TO_UNSIGNED_LONGWORD (X : ADDRESS)     return UNSIGNED_LONGWORD;

function TO_UNSIGNED_LONGWORD (X : AST_HANDLER) return UNSIGNED_LONGWORD;

-- Conventional names for static subtypes of type UNSIGNED_LONGWORD

subtype UNSIGNED_1  is UNSIGNED_LONGWORD range 0 .. 2** 1-1;
subtype UNSIGNED_2  is UNSIGNED_LONGWORD range 0 .. 2** 2-1;
subtype UNSIGNED_3  is UNSIGNED_LONGWORD range 0 .. 2** 3-1;
subtype UNSIGNED_4  is UNSIGNED_LONGWORD range 0 .. 2** 4-1;
subtype UNSIGNED_5  is UNSIGNED_LONGWORD range 0 .. 2** 5-1;

subtype UNSIGNED_6  is UNSIGNED_LONGWORD range 0 .. 2** 6-1;
subtype UNSIGNED_7  is UNSIGNED_LONGWORD range 0 .. 2** 7-1;
subtype UNSIGNED_8  is UNSIGNED_LONGWORD range 0 .. 2** 8-1;
subtype UNSIGNED_9  is UNSIGNED_LONGWORD range 0 .. 2** 9-1;
subtype UNSIGNED_10 is UNSIGNED_LONGWORD range 0 .. 2**10-1;

subtype UNSIGNED_11 is UNSIGNED_LONGWORD range 0 .. 2**11-1;
subtype UNSIGNED_12 is UNSIGNED_LONGWORD range 0 .. 2**12-1;
subtype UNSIGNED_13 is UNSIGNED_LONGWORD range 0 .. 2**13-1;
subtype UNSIGNED_14 is UNSIGNED_LONGWORD range 0 .. 2**14-1;
subtype UNSIGNED_15 is UNSIGNED_LONGWORD range 0 .. 2**15-1;

subtype UNSIGNED_16 is UNSIGNED_LONGWORD range 0 .. 2**16-1;
subtype UNSIGNED_17 is UNSIGNED_LONGWORD range 0 .. 2**17-1;
subtype UNSIGNED_18 is UNSIGNED_LONGWORD range 0 .. 2**18-1;
subtype UNSIGNED_19 is UNSIGNED_LONGWORD range 0 .. 2**19-1;
subtype UNSIGNED_20 is UNSIGNED_LONGWORD range 0 .. 2**20-1;

subtype UNSIGNED_21 is UNSIGNED_LONGWORD range 0 .. 2**21-1;
subtype UNSIGNED_22 is UNSIGNED_LONGWORD range 0 .. 2**22-1;
subtype UNSIGNED_23 is UNSIGNED_LONGWORD range 0 .. 2**23-1;
subtype UNSIGNED_24 is UNSIGNED_LONGWORD range 0 .. 2**24-1;
subtype UNSIGNED_25 is UNSIGNED_LONGWORD range 0 .. 2**25-1;

subtype UNSIGNED_26 is UNSIGNED_LONGWORD range 0 .. 2**26-1;
subtype UNSIGNED_27 is UNSIGNED_LONGWORD range 0 .. 2**27-1;
subtype UNSIGNED_28 is UNSIGNED_LONGWORD range 0 .. 2**28-1;
subtype UNSIGNED_29 is UNSIGNED_LONGWORD range 0 .. 2**29-1;
subtype UNSIGNED_30 is UNSIGNED_LONGWORD range 0 .. 2**30-1;
subtype UNSIGNED_31 is UNSIGNED_LONGWORD range 0 .. 2**31-1;

-- Function for obtaining global symbol values

function IMPORT_VALUE (SYMBOL : STRING) return UNSIGNED_LONGWORD;

-- VAX device and process register operations
```

```
    function  READ_REGISTER (SOURCE : UNSIGNED_BYTE)     return UNSIGNED_BYTE;
    function  READ_REGISTER (SOURCE : UNSIGNED_WORD)     return UNSIGNED_WORD;
    function  READ_REGISTER (SOURCE : UNSIGNED_LONGWORD) return UNSIGNED_LONGWORD;

    procedure WRITE_REGISTER(SOURCE : UNSIGNED_BYTE;
                             TARGET : out UNSIGNED_BYTE);
    procedure WRITE_REGISTER(SOURCE : UNSIGNED_WORD;
                             TARGET : out UNSIGNED_WORD);
    procedure WRITE_REGISTER(SOURCE : UNSIGNED_LONGWORD;
                             TARGET : out UNSIGNED_LONGWORD);

    function  MFPR (REG_NUMBER : INTEGER) return UNSIGNED_LONGWORD;
    procedure MTPR (REG_NUMBER : INTEGER;
                    SOURCE     : UNSIGNED_LONGWORD);

--  VAX interlocked-instruction procedures

    procedure CLEAR_INTERLOCKED (BIT       : in out BOOLEAN;
                                 OLD_VALUE : out BOOLEAN);
    procedure SET_INTERLOCKED   (BIT       : in out BOOLEAN;
                                 OLD_VALUE : out BOOLEAN);

    type ALIGNED_SHORT_INTEGER is
       record
          VALUE : SHORT_INTEGER := 0;
       end record;
    for  ALIGNED_SHORT_INTEGER use
       record
          at mod 2;
       end record;

    procedure ADD_INTERLOCKED (ADDEND : in     SHORT_INTEGER;
                               AUGEND : in out ALIGNED_SHORT_INTEGER;
                               SIGN   : out    INTEGER);

    type INSQ_STATUS is (OK_NOT_FIRST, FAIL_NO_LOCK, OK_FIRST);
    type REMQ_STATUS is (OK_NOT_EMPTY, FAIL_NO_LOCK,
                         OK_EMPTY, FAIL_WAS_EMPTY);

    procedure INSQHI (ITEM   : in  ADDRESS;
                      HEADER : in  ADDRESS;
                      STATUS : out INSQ_STATUS);

    procedure REMQHI (HEADER : in  ADDRESS;
                      ITEM   : out ADDRESS;
                      STATUS : out REMQ_STATUS);

    procedure INSQTI (ITEM   : in  ADDRESS;
                      HEADER : in  ADDRESS;
                      STATUS : out INSQ_STATUS);

    procedure REMQTI (HEADER : in  ADDRESS;
                      ITEM   : out ADDRESS;
                      STATUS : out REMQ_STATUS);
```

```
private
    -- Not shown
end SYSTEM;
```

F.4 Restrictions on Representation Clauses

The representation clauses allowed in VAX Ada are length. enumeration, record representation, and address clauses.

In VAX Ada, a representation clause for a generic formal type or a type that depends on a generic formal type is not allowed. In addition, a representation clause for a composite type that has a component or subcomponent of a generic formal type or a type derived from a generic formal type is not allowed.

F.5 Conventions for Implementation-Generated Names Denoting Implementation-Dependent Components in Record Representation Clauses

VAX Ada does not allocate implementation-dependent components in records.

F.6 Interpretation of Expressions Appearing in Address Clauses

Expressions appearing in address clauses must be of the type ADDRESS defined in the package SYSTEM (see 13.7a.1 and F.3). In VAX Ada, values of type SYSTEM.ADDRESS are interpreted as integers in the range 0..MAX_INT, and they refer to addresses in the user half of the VAX address space.

VAX Ada allows address clauses for variables (see 13.5).

VAX Ada does not support interrupts.

F.7 Restrictions on Unchecked Type Conversions

VAX Ada supports the generic function UNCHECKED_CONVERSION with the restrictions given in section 13.10.2.

F.8 Implementation-Dependent Characteristics of Input-Output Packages

The VAX Ada predefined packages and their operations are implemented using VAX Record Management Services (RMS) file organizations and facilities. To give users the maximum benefit of the underlying VAX RMS input-output facilities, VAX Ada provides packages in addition to the packages SEQUENTIAL_IO, DIRECT_IO, TEXT_IO, and IO_EXCEPTIONS, and VAX Ada accepts VAX RMS File Definition Language (FDL) statements in form strings. The following sections summarize the implementation-dependent characteristics of the VAX Ada input-output packages. The *VAX Ada Run-Time Reference Manual* discusses these characteristics in more detail.

F.8.1 Additional VAX Ada Input-Output Packages

In addition to the language-defined input-output packages (SEQUENTIAL_IO, DIRECT_IO, and TEXT_IO), VAX Ada provides the following input-output packages:

- RELATIVE_IO (see 14.2a.3)
- INDEXED_IO (see 14.2a.5)
- SEQUENTIAL_MIXED_IO (see 14.2b.4)
- DIRECT_MIXED_IO (see 14.2b.6)
- RELATIVE_MIXED_IO (see 14.2b.8)
- INDEXED_MIXED_IO (see 14.2b.10)

VAX Ada does not provide the package LOW_LEVEL_IO.

F.8.2 Auxiliary Input-Output Exceptions

VAX Ada defines the exceptions needed by the packages RELATIVE_IO, INDEXED_IO, RELATIVE_MIXED_IO, and INDEXED_MIXED_IO in the package AUX_IO_EXCEPTIONS (see 14.5a).

F.8.3 Interpretation of the FORM Parameter

The value of the FORM parameter for the OPEN and CREATE procedures of each input-output package may be a string whose value is interpreted as a sequence of statements of the VAX Record Management Services (RMS) File Definition Language (FDL), or it may be a string whose value is interpreted as the name of an external file containing FDL statements.

The use of the FORM parameter is described for each input-output package in chapter 14. For information on the default FORM parameters for each VAX Ada input-output package and for information on using the FORM parameter to specify external file attributes, see the *VAX Ada Run-Time Reference Manual*. For information on FDL, see the *Guide to VAX/VMS File Applications* and the *VAX/VMS File Definition Language Facility Reference Manual.*

F.8.4 Implementation-Dependent Input-Output Error Conditions

As specified in section 14.4, VAX Ada raises the following language-defined exceptions for error conditions occurring during input-output operations: STATUS_ERROR, MODE_ERROR, NAME_ERROR, USE_ERROR, END_ERROR, DATA_ERROR, and LAYOUT_ERROR. In addition, VAX Ada raises the following exceptions for relative and indexed input-output operations: LOCK_ERROR, EXISTENCE_ERROR, and KEY_ERROR. VAX Ada does not raise the language-defined exception DEVICE_ERROR; device-related error conditions cause USE_ERROR to be raised.

The exception USE_ERROR is raised under the following conditions:

- In all CREATE operations if the mode specified is IN_FILE.
- In all CREATE operations if the file attributes specified by the FORM parameter are not supported by the package.

- In the WRITE operations on relative or indexed files if the element in the position indicated has already been written.
- In the UPDATE and DELETE_ELEMENT operations on relative or indexed files if the element to be updated or deleted is not locked.
- In the UPDATE operations on indexed files if the specified key violates the external file attributes.
- In the SET_LINE_LENGTH and SET_PAGE_LENGTH operations on text files if the lengths specified are inappropriate for the external file.
- If the capacity of the external file has been exceeded.

The exception NAME_ERROR is raised as specified in section 14.4: by a call of a CREATE or OPEN procedure if the string given for the NAME parameter does not allow the identification of an external file. In VAX Ada, the value of a NAME parameter can be a string that denotes a VAX/VMS file specification or a VAX/VMS logical name (in either case, the string names an external file). For a CREATE procedure, the value of a NAME parameter can also be a null string, in which case it names a temporary external file that is deleted when the main program exits. The *VAX Ada Run-Time Reference Manual* explains the naming of external files in more detail.

F.9 Other Implementation Characteristics

Implementation characteristics having to do with the definition of a main program, various numeric ranges, and implementation limits are summarized in the following sections.

F.9.1 Definition of a Main Program

A library unit can be used as a main program provided it has no formal parameters and, in the case of a function, if its returned value is a discrete type. If the main program is a procedure, the status returned to the VAX/VMS environment upon normal completion of the procedure is the value one. If the main procedure is a function, the status returned is the function value. Note that when a main function returns a discrete value whose size is less than 32 bits, the value is zero or sign extended as appropriate.

F.9.2 Values of Integer Attributes

The ranges of values for integer types declared in the package STANDARD are as follows:

SHORT_SHORT_INTEGER	-128 .. 127
SHORT_INTEGER	-32768 .. 32767
INTEGER	-2147483648 .. 2147483647

For the packages DIRECT_IO, RELATIVE_IO, SEQUENTIAL_MIXED_IO, DIRECT_MIXED_IO, RELATIVE_MIXED_IO, INDEXED_MIXED_IO, and TEXT_IO, the ranges of values for types COUNT and POSITIVE_COUNT are as follows:

COUNT	0 .. 2147483647
POSITIVE_COUNT	1 .. 2147483647

For the package TEXT_IO, the range of values for the type FIELD is as follows:

FIELD	0 .. 2147483647

F.9.3 Values of Floating Point Attributes

Attribute	F_Floating Value and Approximate Decimal Equivalent
DIGITS	6
MANTISSA	21
EMAX	84
EPSILON	16#0.1000_000#e-4
approximately	9.53674E-07
SMALL	16#0.8000_000#e-21
approximately	2.58494E-26
LARGE	16#0.FFFF_F80#e+21
approximately	1.93428E+25

Attribute	F_Floating Value and Approximate Decimal Equivalent
SAFE_EMAX	127
SAFE_SMALL	16#0.1000_000#e-31
approximately	2.93874E-39
SAFE_LARGE	16#0.7FFF_FC0#e+32
approximately	1.70141E+38
FIRST	-16#0.7FFF_FF8#e+32
approximately	-1.70141E+38
LAST	16#0.7FFF_FF8#e+32
approximately	1.70141E+38
MACHINE_RADIX	2
MACHINE_MANTISSA	24
MACHINE_EMAX	127
MACHINE_EMIN	-127
MACHINE_ROUNDS	True
MACHINE_OVERFLOWS	True

Attribute	D_Floating Value and Approximate Decimal Equivalent
DIGITS	9
MANTISSA	31
EMAX	124
EPSILON	16#0.4000_0000_0000_000#e-7
approximately	9.3132257461548E-10
SMALL	16#0.8000_0000_0000_000#e-31
approximately	2.3509887016446E-38
LARGE	16#0.FFFF_FFFE_0000_000#e+31
approximately	2.1267647922655E+37
SAFE_EMAX	127
SAFE_SMALL	16#0.1000_0000_0000_000#e-31
approximately	2.9387358770557E-39

Attribute	D_Floating Value and Approximate Decimal Equivalent
SAFE_LARGE	16#0.7FFF_FFFF_0000_000#e+32
approximately	1.7014118338124E+38
FIRST	-16#0.7FFF_FFFF_FFFF_FF8#e+32
approximately	-1.7014118346047E+38
LAST	16#0.7FFF_FFFF_FFFF_FF8#e+32
approximately	1.7014118346047E+38
MACHINE_RADIX	2
MACHINE_MANTISSA	56
MACHINE_EMAX	127
MACHINE_EMIN	-127
MACHINE_ROUNDS	True
MACHINE_OVERFLOWS	True

Attribute	G_Floating Value and Approximate Decimal Equivalent
DIGITS	15
MANTISSA	51
EMAX	204
EPSILON	16#0.4000_0000_0000_00#e-12
approximately	8.881784197001E-016
SMALL	16#0.8000_0000_0000_00#e-51
approximately	1.944692274332E-062
LARGE	16#0.FFFF_FFFF_FFFF_E0#e+51
approximately	2.571100870814E+061
SAFE_EMAX	1023
SAFE_SMALL	16#0.1000_0000_0000_00#e-255
approximately	5.562684646268E-309
SAFE_LARGE	16#0.7FFF_FFFF_FFFF_F0#e+256
approximately	8.988465674312E+307

Attribute	G_Floating Value and Approximate Decimal Equivalent
FIRST	-16#0.7FFF_FFFF_FFFF_FC#e+256
approximately	-8.988465674312E+307
LAST	16#0.7FFF_FFFF_FFFF_FC#e+256
approximately	8.988465674312E+307
MACHINE_RADIX	2
MACHINE_MANTISSA	53
MACHINE_EMAX	1023
MACHINE_EMIN	-1023
MACHINE_ROUNDS	True
MACHINE_OVERFLOWS	True

Attribute	H_Floating Value and Approximate Decimal Equivalent
DIGITS	33
MANTISSA	111
EMAX	444
EPSILON	16#0.4000_0000_0000_0000_0000_0000_0000_0#e-27
approximately	7.703719777548943412223911770339 7E-0034
SMALL	16#0.8000_0000_0000_0000_0000_0000_0000_0#e-111
approximately	1.1006568214637918210934318020936E-0134
LARGE	16#0.FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFE_0#e+111
approximately	1.5427420268475430659332737993000E+0133
SAFE_EMAX	16383
SAFE_SMALL	16#0.1000_0000_0000_0000_0000_0000_0000_0#e-4095
approximately	8.4052578577802337656566945433044E-4933
SAFE_LARGE	16#0.7FFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_0#e+4096
approximately	5.9486574767861588254287966331400E+4931
FIRST	-16#0.7FFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_C#e+4096
approximately	-5.9486574767861588254287966331400E+4931

Attribute	H_Floating Value and Approximate Decimal Equivalent
LAST	16#0.7FFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_C#e+4096
approximately	5.948657476786158825428796633140OE+4931
MACHINE_RADIX	2
MACHINE_MANTISSA	113
MACHINE_EMAX	16383
MACHINE_EMIN	-16383
MACHINE_ROUNDS	True
MACHINE_OVERFLOWS	True

F.9.4 Attributes of Type DURATION

The values of the significant attributes of type DURATION are as follows:

DURATION'DELTA	1.00000E-04
DURATION'SMALL	2^{-14}
DURATION'FIRST	-131072.0000
DURATION'LAST	131071.9999
DURATION'LARGE	1.31071999938964843 75E+05

F.9.5 Implementation Limits

Limit	Description
32	Maximum number of formal parameters in a subprogram or entry declaration that are of an unconstrained record type
120	Maximum identifier length (number of characters)
120	Maximum number of characters in a source line
245	Maximum number of discriminants for a record type

Limit	Description
246	Maximum number of formal parameters in an entry or subprogram declaration
255	Maximum number of dimensions in an array type
1023	Maximum number of library units and subunits in a compilation closure[1]
4095	Maximum number of library units and subunits in an execution closure[2]
32757	Maximum number of objects declared with PSECT_OBJECT pragmas
65535	Maximum number of enumeration literals in an enumeration type definition
65535	Maximum number of characters in a value of the predefined type STRING
65535	Maximum number of frames that an exception can propagate
65535	Maximum number of lines in a source file
$2^{31}-1$	Maximum number of bits in any object

[1] The compilation closure of a given unit is the total set of units that the given unit depends on, directly and indirectly.

[2] The execution closure of a given unit is the compilation closure plus all associated secondary units (library bodies and subunits).

APPENDIX C

TEST PARAMETERS

Certain tests in the ACVC make use of implementation-dependent values, such as the maximum length of an input line and invalid file names. A test that makes use of such values is identified by the extension .TST in its file name. Actual values to be substituted are represented by names that begin with a dollar sign. A value must be substituted for each of these names before the test is run. The values used for this validation are given below.

Name and Meaning	Value
$BIG_ID1 Identifier the size of the maximum input line length with varying last character.	119 A's and a '1'
$BIG_ID2 Identifier the size of the maximum input line length with varying last character.	119 A's and a '2'
$BIG_ID3 Identifier the size of the maximum input line length with varying middle character.	119 A's and a '3' in the middle
$BIG_ID4 Identifier the size of the maximum input line length with varying middle character.	119 A's and a '4' in the middle
$BIG_INT_LIT An integer literal of value 298 with enough leading zeroes so that it is the size of the maximum line length.	116 0's and 0298
$BIG_REAL_LIT A universal real literal of value 690.0 with enough leading zeroes to be the size of the maximum line length.	115 0's and 690.0

$BIG_STRING1 A string literal which when catenated with BIG_STRING2 yields the image of BIG_ID1.	"60 A's"
$BIG_STRING2 A string literal which when catenated to the end of BIG_STRING1 yields the image of BIG_ID1.	"59 A's followed by 1"
$BLANKS A sequence of blanks twenty characters less than the size of the maximum line length.	100 blanks
$COUNT_LAST A universal integer literal whose value is TEXT_IO.COUNT'LAST.	2147483647
$FIELD_LAST A universal integer literal whose value is TEXT_IO.FIELD'LAST.	2147483647
$FILE_NAME_WITH_BAD_CHARS An external file name that either contains invalid characters or is too long.	BAD-CHARS^#.%!X
$FILE_NAME_WITH_WILD_CARD_CHAR An external file name that either contains a wild card character or is too long.	WILD-CHAR*.NAM
$GREATER_THAN_DURATION A universal real literal that lies between DURATION'BASE'LAST and DURATION'LAST or any value in the range of DURATION.	75000.0
$GREATER_THAN_DURATION_BASE_LAST A universal real literal that is greater than DURATION'BASE'LAST.	131073.0
$ILLEGAL_EXTERNAL_FILE_NAME1 An external file name which contains invalid characters.	BAD-CHAR @.~!

$ILLEGAL_EXTERNAL_FILE_NAME2 An external file name which is too long.	THIS-FILE-WOULD-BE-PERFECTLY-LEGAL-IF-IT-WERE-NOT-SO-LONG.SO-THERE
$INTEGER_FIRST A universal integer literal whose value is INTEGER'FIRST.	-2147483648
$INTEGER_LAST A universal integer literal whose value is INTEGER'LAST.	2147483647
$INTEGER_LAST_PLUS_1 A universal integer literal whose value is INTEGER'LAST + 1.	2147483648
$LESS_THAN_DURATION A universal real literal that lies between DURATION'BASE'FIRST and DURATION'FIRST or any value in the range of DURATION.	-75000.0
$LESS_THAN_DURATION_BASE_FIRST A universal real literal that is less than DURATION'BASE'FIRST.	-131073.0
$MAX_DIGITS Maximum digits supported for floating-point types.	33
$MAX_IN_LEN Maximum input line length permitted by the implementation.	120
$MAX_INT A universal integer literal whose value is SYSTEM.MAX_INT.	2147483647
$MAX_INT_PLUS_1 A universal integer literal whose value is SYSTEM.MAX_INT+1.	2147483648
$MAX_LEN_INT_BASED_LITERAL A universal integer based literal whose value is 2#11# with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	2: followed by 115 0's followed by 11:

$MAX_LEN_REAL_BASED_LITERAL — 16: followed by 113 0's followed by F.E:
A universal real based literal whose value is 16:F.E: with enough leading zeroes in the mantissa to be MAX_IN_LEN long.

$MAX_STRING_LITERAL — "118 A's"
A string literal of size MAX_IN_LEN, including the quote characters.

$MIN_INT — -2147483648
A universal integer literal whose value is SYSTEM.MIN_ INT.

$NAME — SHORT_SHORT_INTEGER
A name of a predefined numeric type other than FLOAT, INTEGER, SHORT_FLOAT, SHORT_INTEGER, LONG_FLOAT, or LONG_INTEGER.

$NEG_BASED_INT — 16#FFFFFFFE#
A based integer literal whose highest order nonzero bit falls in the sign bit position of the representation for SYSTEM.MAX_INT.

APPENDIX D

WITHDRAWN TESTS

Some tests are withdrawn from the ACVC because they do not conform tc the Ada Standard. The following 25 tests had been withdrawn at the time of validation testing for the reasons indicated. A reference of the form "AI-ddddd" is to an Ada Commentary.

B28003A A basic delcaration (line 36) wrongly follows a later declaration.

E28005C This test requires that 'PRAGMA LIST (ON);' not appear in a listing that has been suspended by a previous "pragma LIST (OFF);"; the Ada Standard is not clear on this point, and the matter will be reviewed by the ALMP.

C34004A The expression in line 168 wrongly yield a value outside of the range of the target type T, raising CONSTRAINT_ERROR.

C35502P The equality operators in lines 62 and 69 should be inequality operators

A35902C Line 17's assignment of the nomimal upper bound of a fixed-point type to an object of that type raises CONSTRAINT_ERROR, for that value lies outside of the actual range of the type.

C35904A The elaboration of the fixed-point subtype on line 28 wrongly raises CONSTRAINT_ERROR, because its upper bound exceeds that of the type.

C35A03E & R These tests assume that attribute 'MANTISSA returns 0 when applied to a fixed-point type with a null range, but the Ada Standard does not support this assumption.

C37213H The subtype declaration of SCONS in line 100 is wrongly expected to raise an exception when elaborated.

C37213J The aggregate in line 451 wrongly raises CONSTRAINT_ERROR.

C37215C,E, G,H Various discriminant constraints are wrongly expected to be incompatible with the type CONS.

C38102C The fixed-point conversion on line 23 wrongly raises CONSTRAINT_ERROR.

C41402A 'STORAGE_SIZE is wrongly applied to an object of an access type.

C45614C REPORT.IDENT_INT has an argument of the wrong type (LONG_INTEGER).

A74106C A bound specified in a fixed-point subtype declaration lies
C8501B outside of that calculated for the base type, raising
C87B04B CONSTRAINT_ERROR. Errors of this sort occur re lines 37 &
CC1311B 59, 142 & 143, 16 & 48, and 252 & 253 of the four tests, respectively (and possibly elsewhere)

BC3105A Lines 159..168 are wrongly expected to be incorrect; they are correct.

AD1A01A The declaration of subtype INT3 raises CONSTRAINT_ERROR for implementations that select INT'SIZE to be 16 or greater.

CE2401H The record aggregates in lines 105 and 117 contain the wrong values.

CE3208A This test expects that an attempt to open the default output file (after it was closed) with MODE_IN file raises NAME_ERROR or USE_ERROR; by commentary AI-00048, MODE_ERROR should be raised.

END
DATE
FILMED
DTIC
11-88